ACCESS15-0017
Oceanographic In-situ Interoperability Project

Deliverable # D5/ OIIP-74

IMPLEMENTATION OF TAGBASE IN POSTGRESQL

Version: # 2
Aug 25, 2018
Revision: # 2

JPL URS CL#:

Acknowledgement: The research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, University Cooperation for Atmospheric Research, and University of Massachusetts-Boston under a contract with the National Aeronautics and Space Administration.

Document Change Record

Author	Update Description	Pages/paragraphs changed	Date of revision
Chi Hin Lam	1. Original v1.0 Draft	All	May 31, 2018
Chi Hin Lam	2. Updated data model	All	Jun 19, 2018
Vardis Tsontos	3. Review, minor edits and addition of Deployment section and Appendix C	ALL	Aug 25, 2018

Table of Contents

List of Figures

List of Tables

1. Overview

Tagbase is a relational database application for the management of biologging data from electronic tags deployed on various marine animals. Tagbase implements a comprehensive relational model handling archival, pop-up archival satellite and telemetry tag files from all major instrument manufacturers. To maximum its value for the community, Tagbase is ported from Microsoft Access into PostgreSQL, a popular open source database management system, as part of the OIIP project. The resulting schema is hosted as open-source code back in Tagbase's Github (https://github.com/tagbase/tagbase). This document describes the design and implementation aspects of Tagbase in PostgreSQL.

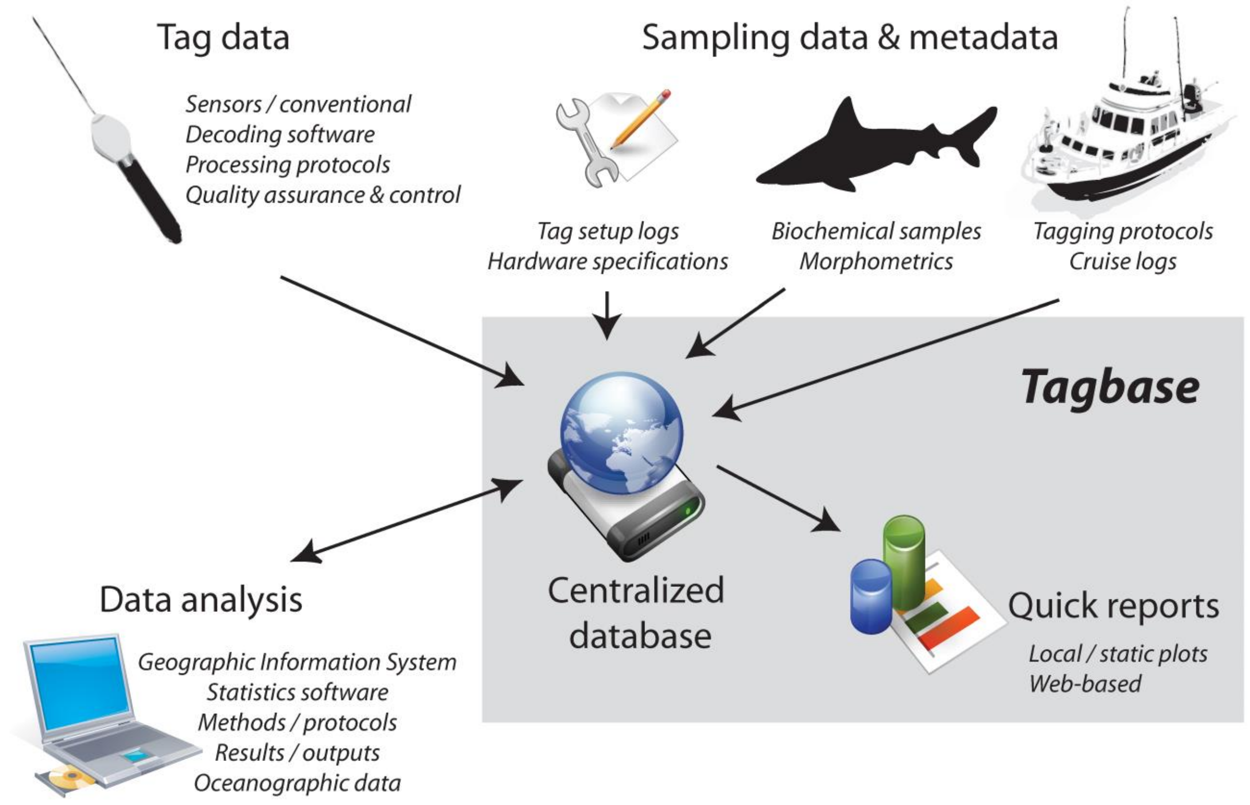


Figure 1 Overall architecture of Tagbase. Lam CH, Tsontos VM (2011) . PLoS ONE 6(7): e21810. doi:10.1371/journal.pone.0021810

2. Review of Tagbase relational data model

Existing relational data model in Tagbase is customized to accommodate various output formats from different manufacturers. While this design is more readily recognized by end users, table structures are hard to evolve through time along with changes made by manufacturers. As manufacturers add new sensors to their product line-ups, a normalized relational model must be developed to achieve (1) flexibility to add new measurement types and values, and (2) consistent table structures that allow access of data by queries, materialized views, and other programs. To meet both requirements, we streamlined Tagbase's existing data model, and implemented a normalized template to house all possible measurements from manufacturers' output files. The results of such are described in the following sections.

3. Electronic Tag Universal File Format (eTUFF)

A key impedance to efficiently handle or import data outputs from different instrument manufacturers is the variety of file structures used. While most output files are formatted as simple text/ ASCII files, they do not share a common file header, which is further complicated by non-standard naming of attributes or header fields. A considerably amount of variability exists even within a single manufacturer's output files, likely because new sensors or firmware are introduced or updated over time. We therefore specify a common file format, termed as electronic tag universal file format (eTUFF), to house data in a normalized fashion that can accommodate data from all possible manufacturer outputs. Ideally, an eTUFF shall be generated at "source", i.e., during parsing of sensor data into output files via manufacturer decoding software, in addition to the output files the software currently generates for end users.

Essentially, eTUFF is a self-described comma-separated text file container for sensor data, consisting of a metadata section and a data section. Comment character in eTUFF is denoted by double forward slashes, `//`. Requirements and recommendations for the metadata are covered in JPL URS CL#: 17-4394[1]. The full listing of metadata is available at https://raw.githubusercontent.com/tagbase/tagbase/master/eTagMetadataInventory.csv. While the relevant attributes and level of detail may vary, the goal of the metadata section is to provide enough information (e.g., species, life stage, length or weight measurements) such that a single eTUFF is self-described and fully contained. Refer to Appendix A for an example of the metadata section.

After the metadata section, the data section allows all types of sensor data to be stored under the following header line:

```
// data:
```

```
// DateTime,VariableID,VariableValue,VariableName,VariableUnits
```

Each attribute is described as follows:

- `DateTime` – formatted as *yyyy-mm-dd hh:mm:ss* according to ISO 8601. By default, this date-time is in Greenwich Mean Time (GMT), per most manufacturer's default time zone setting for the on-board clock. This field can be left empty, as denoted by open and close quotes `""`, for timeless elements such as a depth histogram lower bin value.
- `VariableID, VariableName, VariableUnits` – identifier, name and unit for a measurement, e.g., depth or internal temperature. The full listing of currently identifiable observation types is found https://raw.githubusercontent.com/tagbase/tagbase/master/eTUFF-ObservationTypes.xlsx, and partially illustrated in

An eTUFF file can be parsed by the Rosetta tool (http://rosetta.unidata.ucar.edu) to generate a self-described CF/ACDD compliant netCDF file, following OIIP project enhancements. Both files (eTUFF or Rosetta-processed netCDF) can be ingested into Tagbase PostegreSQL.

Table 1 Example of eTUFF supported observations from sensor measurements in electronic tags.

VariableID	VariableName	VariableSource	VariableUnits	CF_StandardName	CF_CanonicalUnits	Notes
1	datetime	common	time			Date time stamp, usually understood to be in GMT time zone when time zone is not specified. Ideally, format after ISO8601 as yyyy-mm-dd hh:mm:ss
2	longitude	common	degree	longitude	degree_east	Decimical degree
3	latitude	common	degree	latitude	degree_north	Decimical degree
4	pressure	common	psi	sea_water_pressure	dbar	Pressure expressed as pound-force per square inch
5	depth	common	meter	depth	meter	Depth in meter, usually converted from pressure measurements
6	temperature	common	Celsius	sea_water_temperature	K	External sensor temperature, usually water temperature in Celsius
7	internal temperature	common	Celsius			Internal sesnor temperature, usually body cavity temperature in Celsius
8	light	common	unitless			Measured blue or green light on a logarithmic scale. Usually expressed as unitless by Wildlife Computers (light sensor in W cm-2), Lotek and Microwave Telemetry (light sensor in Lux) tags. Also termed as relative light levels.
9	accelX	common	G			Acceleration in the X-axis expressed as gravitational force (1 G ~ 9.8 ms-2)
10	accelY	common	G			Acceleration in the Y-axis expressed as gravitational force (1 G ~ 9.8 ms-2)
11	accelZ	common	G			Acceleration in the Z-axis expressed as gravitational force (1 G ~ 9.8 ms-2)
12	accelM	common	G			Magnitude of acceleration expressed as gravitational force (1 G ~ 9.8 ms-2)
13	accelMdelta	common	G			Change in magnitude of acceleration
14	magX	common	nT			Magnetic field strength in the X-axis as nanotesla
15	magY	common	nT			Magnetic field strength in the Y-axis as nanotesla
16	magZ	common	nT			Magnetic field strength in the Z-axis as nanotesla
17	oxygen	common	ml L-1	mole_concentration_of_dissolved_molecular_oxygen_in_sea_water	mol m-3	Dissolved oxygen expressed in milliter per liter
18	OxySat	common	Percent	fractional_saturation_of_oxygen_in_sea_water	1	Oxygen saturation in percent
19	longitudeError	common	degree			Confidence interval is bound by +/- this number
20	latitudeError	common	degree			Confidence interval is bound by +/- this number

Here are examples how eTUFF can accommodate some of the common data outputs, using the information presented in Table 1.

a) Time series measurements – e.g., depth only

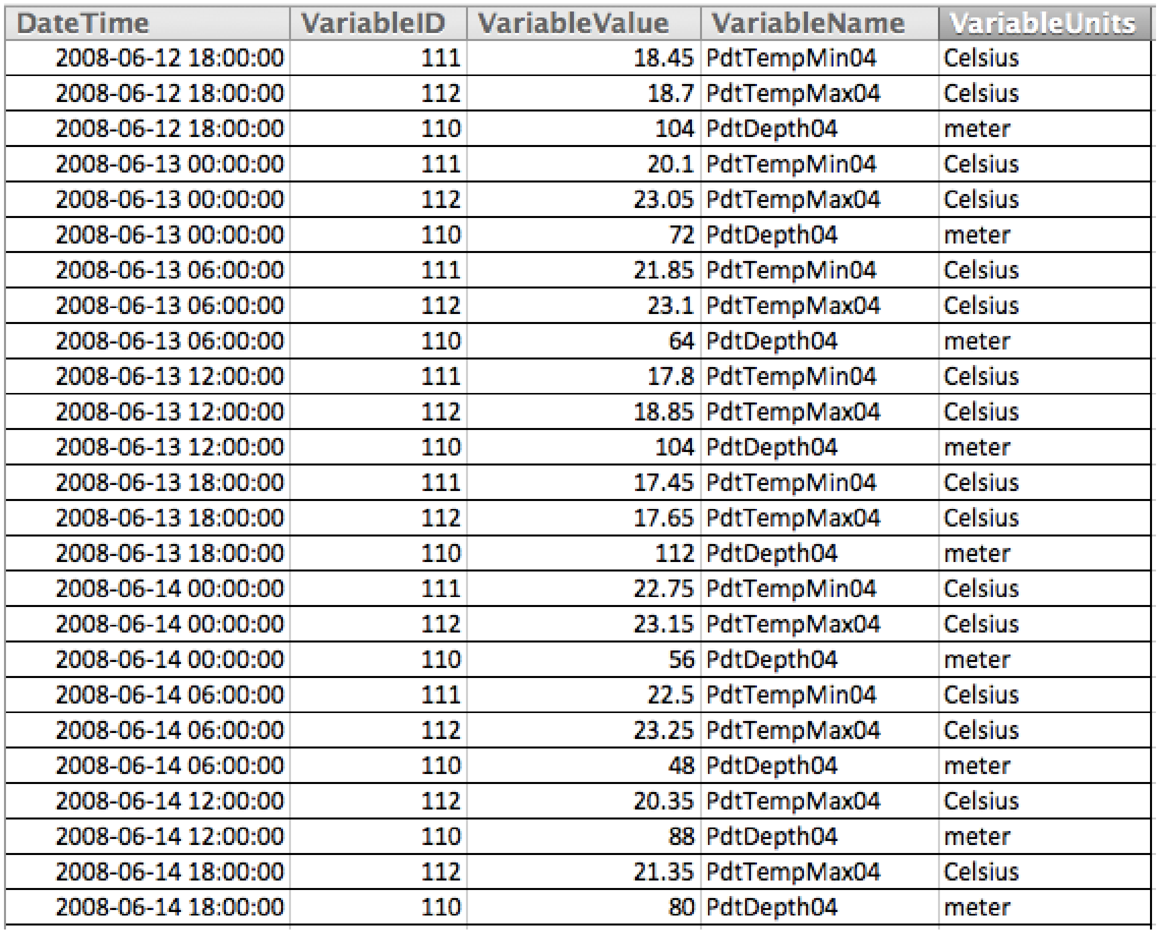
DateTime	VariableID	VariableValue	VariableName	VariableUnits
2007-03-10 21:58:00	5	276.5	depth	meters
2007-03-10 21:59:00	5	265.5	depth	meters
2007-03-10 22:00:00	5	258.5	depth	meters
2007-03-10 22:01:00	5	232	depth	meters
2007-03-10 22:02:00	5	169	depth	meters
2007-03-10 22:03:00	5	141.5	depth	meters
2007-03-10 22:04:00	5	103	depth	meters
2007-03-10 22:05:00	5	25	depth	meters

b) Histogram summaries – e.g., time spent at depth

Histogram usually allows a number of user-specified, pre-programmed bins in which data are summarized into. In the following case, depth bin #8 is between 150 and 200 meters, which are specified by the first two entries. Notice the null values in the *datetime* column since the bin information does not contain any such information.

DateTime	VariableID	VariableValue	VariableName	VariableUnits
	315	150	HistDepthBinMin08	meter
	316	200	HistDepthBinMax08	meter
2005-04-15 21:00:00	372	0.45	TimeAtDepthBin08	proportion
2005-04-16 00:00:00	372	0.2	TimeAtDepthBin08	proportion
2005-04-16 06:00:00	372	0.14	TimeAtDepthBin08	proportion
2005-04-16 12:00:00	372	0.01	TimeAtDepthBin08	proportion
2005-04-16 18:00:00	372	0.88	TimeAtDepthBin08	proportion
2005-04-17 00:00:00	372	0.7	TimeAtDepthBin08	proportion
2005-04-17 06:00:00	372	0.25	TimeAtDepthBin08	proportion
2005-04-17 12:00:00	372	0.4	TimeAtDepthBin08	proportion
2005-04-17 18:00:00	372	0.1	TimeAtDepthBin08	proportion
2005-04-18 00:00:00	372	0.01	TimeAtDepthBin08	proportion

c) Profile measurements – e.g., minimum and maximum temperature at depth measurement #4 (depth measurements are usually picked dynamically by on-board algorithm).

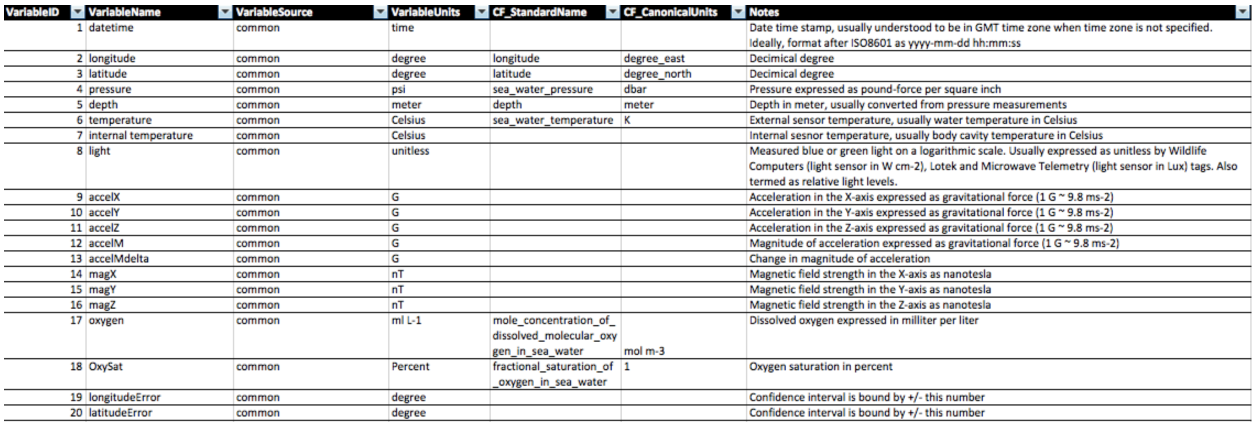


DateTime	VariableID	VariableValue	VariableName	VariableUnits
2008-06-12 18:00:00	111	18.45	PdtTempMin04	Celsius
2008-06-12 18:00:00	112	18.7	PdtTempMax04	Celsius
2008-06-12 18:00:00	110	104	PdtDepth04	meter
2008-06-13 00:00:00	111	20.1	PdtTempMin04	Celsius
2008-06-13 00:00:00	112	23.05	PdtTempMax04	Celsius
2008-06-13 00:00:00	110	72	PdtDepth04	meter
2008-06-13 06:00:00	111	21.85	PdtTempMin04	Celsius
2008-06-13 06:00:00	112	23.1	PdtTempMax04	Celsius
2008-06-13 06:00:00	110	64	PdtDepth04	meter
2008-06-13 12:00:00	111	17.8	PdtTempMin04	Celsius
2008-06-13 12:00:00	112	18.85	PdtTempMax04	Celsius
2008-06-13 12:00:00	110	104	PdtDepth04	meter
2008-06-13 18:00:00	111	17.45	PdtTempMin04	Celsius
2008-06-13 18:00:00	112	17.65	PdtTempMax04	Celsius
2008-06-13 18:00:00	110	112	PdtDepth04	meter
2008-06-14 00:00:00	111	22.75	PdtTempMin04	Celsius
2008-06-14 00:00:00	112	23.15	PdtTempMax04	Celsius
2008-06-14 00:00:00	110	56	PdtDepth04	meter
2008-06-14 06:00:00	111	22.5	PdtTempMin04	Celsius
2008-06-14 06:00:00	112	23.25	PdtTempMax04	Celsius
2008-06-14 06:00:00	110	48	PdtDepth04	meter
2008-06-14 12:00:00	112	20.35	PdtTempMax04	Celsius
2008-06-14 12:00:00	110	88	PdtDepth04	meter
2008-06-14 18:00:00	112	21.35	PdtTempMax04	Celsius
2008-06-14 18:00:00	110	80	PdtDepth04	meter

- . This listing is expected to be updated when manufacturers continue to provide new sensor measurements. Ideally, this list will be vetted by the rest of the tagging community to standardize the vocabulary to describe manufacturer-specific measurements in an easily understandable manner.
- `VariableValue` – the measurement value for a particular sensor, or pre-programmed setting. This value should be numeric, as “double” precision value.

An eTUFF file can be parsed by the Rosetta tool (http://rosetta.unidata.ucar.edu) to generate a self-described CF/ACDD compliant netCDF file, following OIIP project enhancements. Both files (eTUFF or Rosetta-processed netCDF) can be ingested into Tagbase PostegreSQL.

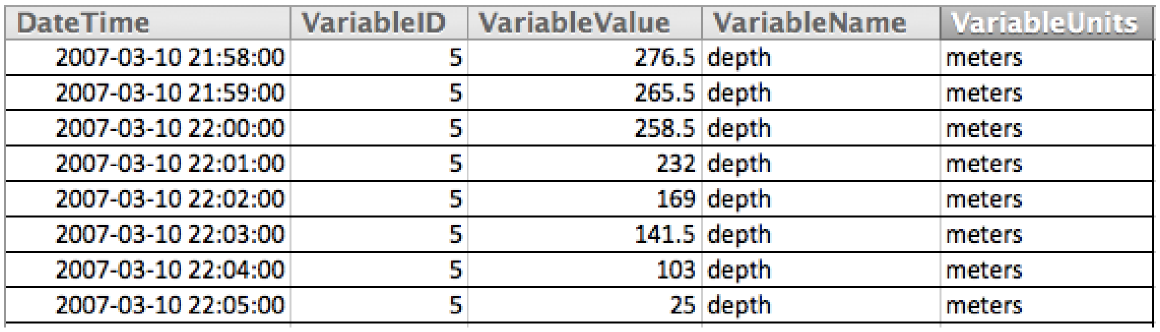
Table 1 Example of eTUFF supported observations from sensor measurements in electronic tags.



VariableID	VariableName	VariableSource	VariableUnits	CF_StandardName	CF_CanonicalUnits	Notes
1	datetime	common	time			Date time stamp, usually understood to be in GMT time zone when time zone is not specified. Ideally, format after ISO8601 as yyyy-mm-dd hh:mm:ss
2	longitude	common	degree	longitude	degree_east	Decimical degree
3	latitude	common	degree	latitude	degree_north	Decimical degree
4	pressure	common	psi	sea_water_pressure	dbar	Pressure expressed as pound-force per square inch
5	depth	common	meter	depth	meter	Depth in meter, usually converted from pressure measurements
6	temperature	common	Celsius	sea_water_temperature	K	External sensor temperature, usually water temperature in Celsius
7	internal temperature	common	Celsius			Internal sesnor temperature, usually body cavity temperature in Celsius
8	light	common	unitless			Measured blue or green light on a logarithmic scale. Usually expressed as unitless by Wildlife Computers (light sensor in W cm-2), Lotek and Microwave Telemetry (light sensor in Lux) tags. Also termed as relative light levels.
9	accelX	common	G			Acceleration in the X-axis expressed as gravitational force (1 G ~ 9.8 ms-2)
10	accelY	common	G			Acceleration in the Y-axis expressed as gravitational force (1 G ~ 9.8 ms-2)
11	accelZ	common	G			Acceleration in the Z-axis expressed as gravitational force (1 G ~ 9.8 ms-2)
12	accelM	common	G			Magnitude of acceleration expressed as gravitational force (1 G ~ 9.8 ms-2)
13	accelMdelta	common	G			Change in magnitude of acceleration
14	magX	common	nT			Magnetic field strength in the X-axis as nanotesla
15	magY	common	nT			Magnetic field strength in the Y-axis as nanotesla
16	magZ	common	nT			Magnetic field strength in the Z-axis as nanotesla
17	oxygen	common	ml L-1	mole_concentration_of_dissolved_molecular_oxygen_in_sea_water	mol m-3	Dissolved oxygen expressed in milliter per liter
18	OxySat	common	Percent	fractional_saturation_of_oxygen_in_sea_water	1	Oxygen saturation in percent
19	longitudeError	common	degree			Confidence interval is bound by +/- this number
20	latitudeError	common	degree			Confidence interval is bound by +/- this number

Here are examples how eTUFF can accommodate some of the common data outputs, using the information presented in Table 1.

d) Time series measurements – e.g., depth only



DateTime	VariableID	VariableValue	VariableName	VariableUnits
2007-03-10 21:58:00	5	276.5	depth	meters
2007-03-10 21:59:00	5	265.5	depth	meters
2007-03-10 22:00:00	5	258.5	depth	meters
2007-03-10 22:01:00	5	232	depth	meters
2007-03-10 22:02:00	5	169	depth	meters
2007-03-10 22:03:00	5	141.5	depth	meters
2007-03-10 22:04:00	5	103	depth	meters
2007-03-10 22:05:00	5	25	depth	meters

e) Histogram summaries – e.g., time spent at depth

Histogram usually allows a number of user-specified, pre-programmed bins in which data are summarized into. In the following case, depth bin #8 is between 150 and 200 meters, which are specified by the first two entries. Notice the null values in the *datetime* column since the bin information does not contain any such information.

DateTime	VariableID	VariableValue	VariableName	VariableUnits
	315	150	HistDepthBinMin08	meter
	316	200	HistDepthBinMax08	meter
2005-04-15 21:00:00	372	0.45	TimeAtDepthBin08	proportion
2005-04-16 00:00:00	372	0.2	TimeAtDepthBin08	proportion
2005-04-16 06:00:00	372	0.14	TimeAtDepthBin08	proportion
2005-04-16 12:00:00	372	0.01	TimeAtDepthBin08	proportion
2005-04-16 18:00:00	372	0.88	TimeAtDepthBin08	proportion
2005-04-17 00:00:00	372	0.7	TimeAtDepthBin08	proportion
2005-04-17 06:00:00	372	0.25	TimeAtDepthBin08	proportion
2005-04-17 12:00:00	372	0.4	TimeAtDepthBin08	proportion
2005-04-17 18:00:00	372	0.1	TimeAtDepthBin08	proportion
2005-04-18 00:00:00	372	0.01	TimeAtDepthBin08	proportion

f) Profile measurements – e.g., minimum and maximum temperature at depth measurement #4 (depth measurements are usually picked dynamically by on-board algorithm).

DateTime	VariableID	VariableValue	VariableName	VariableUnits
2008-06-12 18:00:00	111	18.45	PdtTempMin04	Celsius
2008-06-12 18:00:00	112	18.7	PdtTempMax04	Celsius
2008-06-12 18:00:00	110	104	PdtDepth04	meter
2008-06-13 00:00:00	111	20.1	PdtTempMin04	Celsius
2008-06-13 00:00:00	112	23.05	PdtTempMax04	Celsius
2008-06-13 00:00:00	110	72	PdtDepth04	meter
2008-06-13 06:00:00	111	21.85	PdtTempMin04	Celsius
2008-06-13 06:00:00	112	23.1	PdtTempMax04	Celsius
2008-06-13 06:00:00	110	64	PdtDepth04	meter
2008-06-13 12:00:00	111	17.8	PdtTempMin04	Celsius
2008-06-13 12:00:00	112	18.85	PdtTempMax04	Celsius
2008-06-13 12:00:00	110	104	PdtDepth04	meter
2008-06-13 18:00:00	111	17.45	PdtTempMin04	Celsius
2008-06-13 18:00:00	112	17.65	PdtTempMax04	Celsius
2008-06-13 18:00:00	110	112	PdtDepth04	meter
2008-06-14 00:00:00	111	22.75	PdtTempMin04	Celsius
2008-06-14 00:00:00	112	23.15	PdtTempMax04	Celsius
2008-06-14 00:00:00	110	56	PdtDepth04	meter
2008-06-14 06:00:00	111	22.5	PdtTempMin04	Celsius
2008-06-14 06:00:00	112	23.25	PdtTempMax04	Celsius
2008-06-14 06:00:00	110	48	PdtDepth04	meter
2008-06-14 12:00:00	112	20.35	PdtTempMax04	Celsius
2008-06-14 12:00:00	110	88	PdtDepth04	meter
2008-06-14 18:00:00	112	21.35	PdtTempMax04	Celsius
2008-06-14 18:00:00	110	80	PdtDepth04	meter

4. Tagbase PostgreSQL relational data model

Assuming eTUFF as the input source (described in previous section), a data model is designed for Tagbase in PostgreSQL (Figure 2; Appendix B). The schema can be accessed via this script at https://raw.githubusercontent.com/tagbase/tagbase/master/Tagbase_schema.sql. With this script one should be able to execute the script and generate a virgin database from scratch with all the necessary code table information included as necessary for the eTUFF data ingest scripts to function. The primary keys, relationships, and data types are explicitly described within the script, and therefore, will not be covered here.

Date time information is represented as *yyyy-mm-dd hh:mm:ss* following ISO 8601. Time zone can be specified in the imported data. Refer to https://www.postgresql.org/docs/8.2/static/datatype-datetime.html for details.

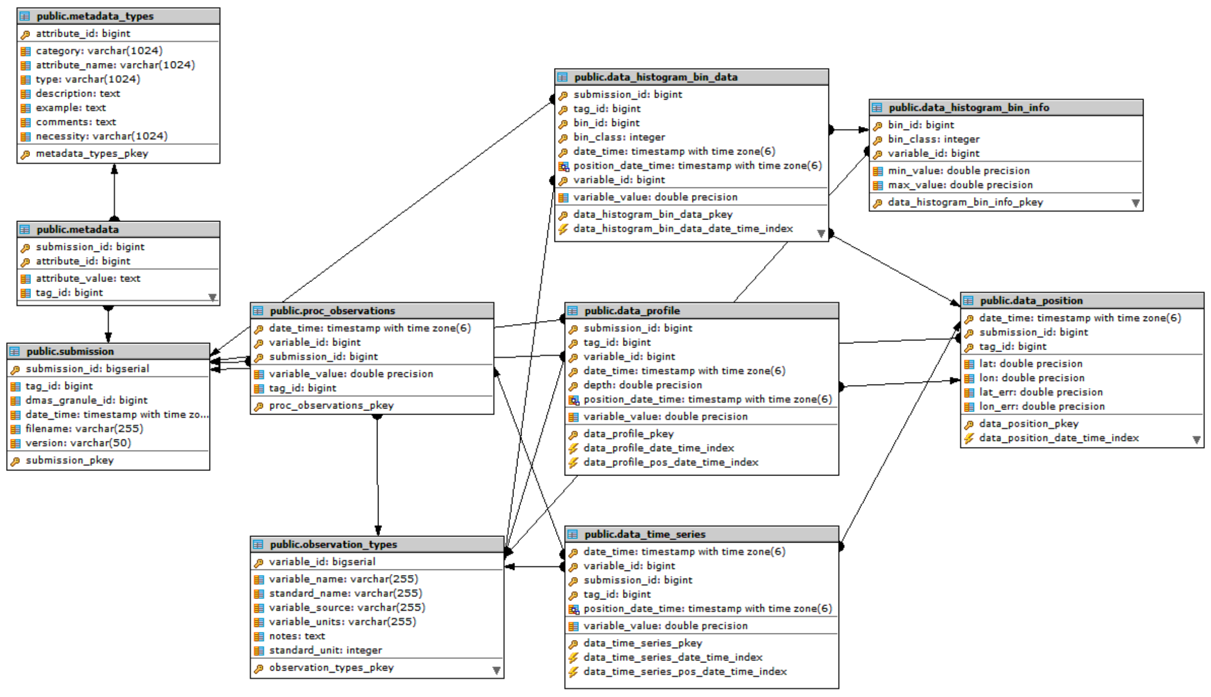


Figure 2 Tagbase PostgreSQL schema.

Architecture and data flow

Tagbase contains 1 housekeeping table, 1 metadata table, 3 lookup tables, and 5 observational data tables. When a dataset in the eTUFF format is being imported, or reimported (for example, in the case of an updated dataset), this event is treated as a new submission, triggering the generation of a unique *submission_id* in the table, `submission`. If the dataset is returned by a tag deployment that does not currently exist in the database, a new *tag_id* is also generated in the table, `submission`. To determine whether a dataset is being reimported i.e., looking up a previously generated *tag_id*, we check for the required metadata attribute, *instrument_name*, in the eTUFF metadata section, which is an identifier that is unique within the data provider/ research's own organization.

The *submission_id* and *tag_id* provide the linkage among various tables, except in the lookup tables. Lookup tables (`metadata_types`, `observation_types`, `histogram_bin_info`) share common information that is independent of a specific tag or submission. For example, the same depth histogram bins can be used by multiple tags deployed across the years for ease of analysis and consistent experimental design.

With the generated *submission_id* and (generated/ looked up) *tag_id*, metadata section of the imported eTUFF file is parsed into the table metadata, and data section into the table proc_observations. Relevant lookup and updates are then performed to obtain the values contained within metadata_types and observation_types tables.

Data sitting in the table proc_observations can be diverse, some of which are specific to particular tag models or manufacturers depending on the data products they provide. The full listing of currently identifiable observation types is found https://raw.githubusercontent.com/tagbase/tagbase/master/eTUFF-ObservationTypes.xlsx. The same listing is housed in the table observation_types.

Four common data representations are shared among outputs from different manufacturers, in which the following tables are designed around, with underlying philosophy that each of these tables contains some unique dimensions (identified below in parentheses) for the data:

1. data_time_series: (time); time here is usually at a high frequency, it can be in intervals of seconds or less
2. data_position: (time, longitude, latitude)
3. data_profile: (time, depth) where depth represents a standard depth in meters
4. data_histogram_bin_data: (time, x, y) where dimension x comes from histogram_bin_info (i.e., how the histogram should be set up), and y comes from a group-by method on a measurement, e.g., count of occurrence or average of temperature. Naturally, the measurement specified by x and y should be meaningful, as there is no point to summarized maximum depth within the depth bin of 0-100 m.

Data sitting in the table proc_observations that conform to one of the above data representations are then relocated (via a set of queries) to the respective data_ tables. The rationale to relocate is to make the data access quicker and more human-readable. Data that do not conform to such representations remain in proc_observations table. This is the last step in the data import process.

A quick breakdown of what each table does follows:

Housekeeping table

- submission – documents the import of a new dataset, including date-time, filename and version of the import. *dmas_granule_id* is used internally by the OIIP project.

Metadata tables

- metadata, metadata_types – contain metadata variables and values listed in https://raw.githubusercontent.com/tagbase/tagbase/master/eTagMetadataInventory.csv

Tables to support observational data

- observation_types – a lookup table that describes the types of measurements/ observations (name, units etc.) The full listing of currently identifiable observation types is found https://raw.githubusercontent.com/tagbase/tagbase/master/eTUFF-ObservationTypes.xlsx.
- proc_observations – contains decoded measurements by sensors on an instrument. This table is the destination in which data are first imported and stored.

Specialized tables for representing observational data: These tables contain data that are relocated from `proc_observations` after being first imported into the database. They hold data that have a distinctive geometry, time series or summarized format.

- `data_position` – contains positional data (*`lat`*, *`lon`*), and their associated errors (*`lat_err`*, *`lon_error`*) in decimal degree
- `data_timeseries` – contains time series (*`date_time`*) observational data (*`variable_id`*, *`variable_value`*). *`variable_id`* links back to the lookup table `observation_types`.
- `data_profile` – contains profile-like observational data (*`variable_id`*, *`variable_value`*) at particular depths (`depth`). *`variable_id`* links back to the lookup table `observation_types`. An example is the mean temperature (defined in `observation_types`) recorded at depths of 0,100, 200, 400 and 800 meters.
- `data_histogram_bin_info` – a lookup table contains metadata on how bins are set up for histograms or summarizing data within an interval of values (e.g., 0-100 meters). *`bin_id`* specifies a particular binning scheme. *`bin_class`* specifies a particular bin (out of a total number of bins) in which the range of values (*`min_value, max_value`*) is set for a type of observation (*`variable_id`* that links back to the lookup table `observation_types`).
- `data_histogram_bin_data` – contains histogram-like data (*`variable_id`*, *`variable_value`*). Works in combination with `data_histogram_bin_info` that describes how binning was set up.

The use of tables, `data_histogram_bin_info` and `data_histogram_bin_data`, can be illustrated with the following pseudo-code:

`data_histogram_bin_info`
`bin_id = 1; 0-100, 100-200, 200-300 meters` depth binning

`observation_types`
`variable_id = 1001; variable_name = frequency` for occurrence expressed as a fraction (0-1), which is the equivalent of time-spent-at-depth or temperature over a temporal period (e.g., a 8-hour summarizing interval. Note this summarizing period should be specified in the metadata, or calculated *post hoc* by taking the difference between two date-time stamps).
`variable_id =  1017; variable_name = tempMean` for mean temperature in Celsius

`data_histogram_bin_data`
```
variable_id = 1001; variable_value = 0.6, 0.3, 0.1
variable_id = 1017; variable_value =  30, 20, 15
```

5. Tagbase Package Installation

The Tagbase distribution delivered by the OIIP project is comprised of several elements conveniently packaged for effortless deployment in a Docker image but also available as individual components for custom/manual installations:

- PostgreSQL implementation of the revised Tagbase relational database schema providing also eTUFF support
- Series of SQL scripts and Python 3 routines facilitating ingestion of electronic tagging data into the Tagbase Postgresql and used also used to automate the creation of standardized outputs in the form of materialized views.
- Tagbase-server: a Rest-API and browser-based form interface allowing for either automated/script based ingestion of tag data files or interactive imports
- PostgreSQL 10 with the PgAdmin 4 tool and all necessary Python 3 libraries
- Docker image packing installation of all of the above components in the correct sequence to initiate a fully-working instance of Tagbase and its server ready for usage.

Tagbase Deployment via Docker

Note that this by far the simplest method for deploying and getting up and running with Tagbase. It is strongly recommended for most users. A video tutorial on the Docker Tagbase deployment is available at https://www.youtube.com/channel/UC8gdz1fOVGndTJkz88Fi7mg

Overview
Tagbase is a Flask application which provides HTTP endpoints for ingestion of
various files into the Tagbase SQL database.

Running with Docker:

Introduction
Docker enables rapid simplified deployment of Tagbase by removing
all services setup and configuration e.g. PostgreSQL, tagbase-server, etc.
This is achieved via [Docker Compose]](https://docs.docker.com/compose/overview/); a tool for defining and
running multi-container Docker applications.
See below for prerequisite installation requirements.

Prerequisites

- Git
- Docker

Either download tabase-server OR clone the source code with Git

```
$ git clone https://${urs_username}@git.earthdata.nasa.gov/scm/oiip/tagbase-server.git
```

N.B. you should replace ${urs_username} with your URS username.

Either way, once you've acquired the tagbase-server source code on your workstation, you need to navigate to the source root directory e.g.

```
$ cd tagbase-server
```

Deployment

N.B. Due to the size of the input datasets we ingest into tagbase-server, it is essential that the container running the service has sufficient available memory (4GB should do the trick).

See this for Mac:
https://docs.docker.com/docker-for-mac/#memory
MEMORY By default, Docker for Mac is set to use 2 GB runtime memory, allocated from the total available memory on your Mac. You can increase the RAM on the app to get faster performance by setting this number higher (for example to 3) or lower (to 1) if you want Docker for Mac to use less memory.

For Windows:
https://docs.docker.com/docker-for-windows/#advanced
Memory - Change the amount of memory the Docker for Windows Linux VM uses

Once sufficient memory is available, to orchestrate and deploy the Tagbase services execute the following from this root directory:

```
$ docker-compose build
$ docker-compose up
```

After a short while, you will now have a completely Dockerized deployment of Tagbase (master), PosgreSQL 10.X and pgAdmin.

See below for accessing the Web Applications.

To stop the docker-compose deployment, simply open a new terminal, navigate to the tagbase-server root directory and execute

```
$ docker-compose stop
```

You will see the services graciously shutdown.

Tagbase Server

N.B. The URI's below may alternate between ***localhost*** and ***0.0.0.0*** depending on whether your workstation is Windows (localhost) or Linux/Mac (0.0.0.0)

Navigate to http://localhost:5433/v1/tagbase/ui/
to see the tagbase-server UI running.

It will really help for you to read the API documentation provided in the Web Application.

Using the eTUFF API, you can execute the following commands to initiate a primitive test ingestion of some sample eTUFF-sailfish-117259.txt data present on the server.

Using curl...

```
curl -X GET --header 'Accept: application/json'
'http://0.0.0.0:5433/v1/tagbase/ingest/etuff?granule_id=1234&file=file%3A%2F%2F%2Fusr%2Fsrc%2Fa
pp%2Fdata%2FeTUFF-sailfish-117259.txt'
```

...or using a Request URL
http://0.0.0.0:5433/v1/tagbase/ingest/etuff?granule_id=1234&file=file%3A%2F%2F%2Fusr%2Fsrc%2Fapp%2Fdata%2FeTUFF-sailfish-117259.txt

N.B. The REST server is capable of ingesting data from many sources e.g. file, ftp, http and https.

pgAdmin

Navigate to http://0.0.0.0:5434/browser/ or on Windows machines http://localhost:5434/browser/ and enter ..

username *tagbase*
password *tagbase*

NB. As PostgreSQL administrator you will be able to change the tagbase user account login account as necessary

You can now:

- Add New Server
- General Tab --> name: tagbase
- Connection Tab --> Host name/address: postgres
- Connection Tab --> Port: 5432
- Connection Tab --> Maintenance database: postgres
- Connection Tab --> Username: tagbase

On the left hand side navigation panel, you will now see the persistent connection to the tagbase database.

Manual Tagbase Deployment [Advanced Users]

Manual installation of Tagbase is described here, however, we recommend this only for system administrators or other more advanced users.

Introduction

The primary deployment strategy for tagbase-server is via Docker. This is explained in the previous section.

For developers who wish to prototype tagbase-server for local deployment, this document provides a HOW_TO.

Requirements

1. Python 3.5.2+
2. Postgres (ensure that the both ```log_timezone = 'UTC'``` and ```timezone = 'UTC'``` are set in ```postgresql.conf```

Tagbase Server

To run the server, execute the following from the root directory:

```
pip3 install -r requirements.txt
python3 -m swagger_server
```

and open your browser to http://localhost:5433/v1/tagbase/ui/
The Swagger API definition lives at http://localhost:5433/v1/tagbase/swagger.json

To edit the Swagger definition file, navigate to http://editor2.swagger.io/#, you can then load the swagger definition and hack away!

PostgreSQL

The Tagbase server requires access to a (Postgres) SQL DB. We can create this as follows
brew install postgresql

You can then start this in the foreground as follows ..

```
postgres -D /usr/local/var/postgres
```

Postgres data tables still need to be defined however. Simply execute …

```
psql -f sqldb/tagbase-schema.sql
```

The above creates the original database, tables, sequences and indexes. You will see a lot of output to the terminal indicating that the database table structures are being loaded and data is being populated.

You can go ahead and now connect to the DB (using ```psql```) and query data e.g.

```
lmcgibbn=# \connect tagbase

You are now connected to database "tagbase" as user "...".

Tagbase=# \dt

List of relations
Schema | Name | Type | Owner
--------+------------------------+-------+----------
public | data_histogram_bin_data | table | lmcgibbn
public | data_histogram_bin_info | table | lmcgibbn
public | data_position | table | lmcgibbn
public | data_profile | table | lmcgibbn
public | data_time_series | table | lmcgibbn
```

```
public | metadata | table | lmcgibbn
public | metadata_types | table | lmcgibbn
public | observation_types | table | lmcgibbn
public | proc_observations | table | lmcgibbn
public | submission | table | lmcgibbn
(11 rows)
tagbase=# SELECT * FROM metadata_types ;
...
attribute_id | category | attribute_name | type | description | example | comments | necessity
--------------+---------------------+--------------------------------+--------+-------------------------------------------------------
1 | instrument | instrument_name | string |

Append an identifer that is unique within your organization. This is essential if a device is recycled. |
16P0100-Refurb2
```

You are now ready to begin interacting with the Tagbase server via the REST endpoints. You can do this by navigating to your Browser at http://localhost:5433/v1/tagbase/ui/ and populating data into the **etuff** endpoint.

6. Generating Tagbase Materialized Views

PostgreSQL Materialized Views extend the concept of database views; virtual tables which represent data of the underlying tables, to the next level that allows views to store data physically, and we call those views materialized views. A materialized view caches the result of a complex expensive query and then allows you to refresh this result periodically.

Upon successful ingestion of files into Tagbase, you are likely to want to generate materialized views in order to access the 'application ready' tagbase data.

N.B. Previously, it was necessary to execute a data migration command which essentially populated initial staging data around the DB. This is now managed by a trigger such that all we need to worry about it generating materialized views.

You can generate the Tagbase materialized views by simply opening the following file (see also Appendix C)

$ open tagbase-server/sql/tagbase-materialized-views.sql

... and executing the contents as a query within the PostgreSQL PgAqmin Query Tool.

Or you if you prefer to use the PostgreSQL command line, generate the Tagbase materialized views by executing the following …

psql -f tagbase-materialized-views.sql

Note that similar to the ingestion and migration routines, generation of the materialized views may take a while so be patient. Once it has completed however, you can browse the materialized views.

N.B. It should be noted that materialized views can only be generated once... this process should not be executed every time a file is ingested!

Development, Support and Community
Please reach out to the OIIP project team at oiip@jpl.nasa.gov

Appendix A – Example of an eTUFF file for time series data recorded by a sailfish

Note most data are truncated for simplicity and clarify

```
// global attributes:
  :institution = "LPRC"
  :references = "Scientific Reports volume 6, Article number: 38163 (2016)
doi:10.1038/srep38163"
// etag instrument attributes:
  :person_owner = "Molly Lutcavage"
  :owner_contact = "melutcavage@gmail.com"
  :device_type = "PSAT"
  :manufacturer = "Microwave Telemetry"
  :model = "X-Tag"
  :serial_number = "20555"
  :ptt = "117259"
// etag attachment attributes:
  :attachment_method = "anchor"
// etag deployment attributes:
  :datetime_release = "2013-04-13"
  :lon_release = "-86.60"
  :lat_release = "21.38"
// etag end of mission attributes:
  :end_type = "recaptured"
  :end_details = "recovered by fishing fleet"
  :date_end = "2013-04-28"
  :lon_end = "-84.93"
  :lat_end = "25.26"
// etag animal attributes:
  :species_capture = "Istiophorus platypterus"
  :length_capture = "171"
  :length_unit_capture = "cm"
  :length_type_capture = "lower jaw fork length"
  :length_method_capture = "measured"
// etag waypoints attributes:
  :waypoints_source = "modeled"
  :waypoints_method = "ukfsst"
// etag quality attributes:
  :found_problem: "no"
  :person_qc: "Tim Lam"
//file attributes:
  :parsing_software: "Tagbase 4.9"
  :schema_observationtypes:
"https://raw.githubusercontent.com/tagbase/tagbase/master/ObservationTypes.xml"
// data:
// DateTime,VariableID,VariableValue,VariableName,VariableUnits
2013-04-14 00:00:00,2,273.40,longitude,degree
2013-04-14 00:00:00,3,21.38,latitude,degree
2013-04-14 00:00:00,19,0.00,longitudeError,degree
2013-04-14 00:00:00,20,0.00,latitudeError,degree
2013-04-14 16:15:00,5,0.00,depth,meters
2013-04-14 16:15:00,6,30.09,temperature,Celsius
2013-04-14 16:15:00,8,4091.00,light,units
2013-04-14 16:17:00,5,0.00,depth,meters
2013-04-14 16:17:00,6,30.09,temperature,Celsius
2013-04-14 16:17:00,8,4091.00,light,units
2013-04-14 16:19:00,5,0.00,depth,meters
2013-04-14 16:19:00,6,30.09,temperature,Celsius
2013-04-14 16:19:00,8,4091.00,light,units
2013-04-14 16:21:00,5,0.00,depth,meters
2013-04-14 16:21:00,6,30.09,temperature,Celsius
2013-04-14 16:21:00,8,4091.00,light,units
2013-04-14 16:23:00,5,0.00,depth,meters
```

Appendix B – Table structures of Tagbase in PostgreSQL

```
CREATE TABLE submission
(
    submission_id   BIGSERIAL                     PRIMARY KEY,
    tag_id          bigint                        NOT NULL,
    dmas_granule_id bigint,
    date_time       timestamp(6) with time zone   DEFAULT current_timestamp,
    filename        character varying(255),
    version         character varying(50)
);

CREATE TABLE observation_types
(
    variable_id     BIGSERIAL               PRIMARY KEY,
    variable_name   character varying(255)  UNIQUE NOT NULL,
    standard_name   character varying(255),
    variable_source character varying(255),
    variable_units  character varying(255),
    notes           text
);

CREATE TABLE proc_observations
(
    date_time           timestamp(6) with time zone,
    variable_id         bigint                        NOT NULL  REFERENCES
observation_types (variable_id),
    variable_value      double precision              NOT NULL,
    submission_id       bigint                        NOT NULL,
    tag_id              bigint                        NOT NULL,
  REFERENCES submission (submission_id) ON DELETE CASCADE
);

CREATE TABLE metadata_types
(
   attribute_id     bigint                      PRIMARY KEY,
   category         character varying(1024)   NOT NULL,
   attribute_name   character varying(1024)   NOT NULL,
   type             character varying(1024)   NOT NULL,
   description      text                      NOT NULL,
   example          text,
   comments         text,
   necessity        character varying(1024)  NOT NULL
);

CREATE TABLE metadata
(
    submission_id     bigint    NOT NULL  REFERENCES submission (submission_id) ON
DELETE CASCADE,
    attribute_id      bigint    NOT NULL  REFERENCES metadata_types (attribute_id),
    attribute_value   text      NOT NULL
);

CREATE TABLE data_time_series
(
    date_time           timestamp(6) with time zone,
    variable_id         bigint                        NOT NULL  REFERENCES
observation_types (variable_id),
    variable_value      double precision              NOT NULL,
    submission_id       bigint                        NOT NULL  REFERENCES submission
(submission_id) ON DELETE CASCADE,
    tag_id              bigint                        NOT NULL,
    position_date_time  timestamp(6) with time zone
);

CREATE TABLE data_position
(
    date_time           timestamp(6) with time zone,
    lat                 double precision,
    lon                 double precision,
    lat_err             double precision,
    lon_err             double precision,
    submission_id       bigint                        NOT NULL  REFERENCES submission
(submission_id) ON DELETE CASCADE,
    tag_id              bigint                        NOT NULL
);


CREATE TABLE data_histogram_bin_info
(
    variable_id         bigint                        NOT NULL  REFERENCES
observation_types (variable_id),
    bin_id          bigint                        NOT NULL  ,
    bin_class       integer                       NOT NULL,
    min_value       double precision,
    max_value       double precision,
    UNIQUE (bin_id, bin_class)
);

CREATE TABLE data_histogram_bin_data
(
    submission_id     bigint    NOT NULL  REFERENCES submission (submission_id) ON
DELETE CASCADE,
    tag_id            bigint    NOT NULL,
    bin_id            bigint    NOT NULL  REFERENCES data_histogram_bin_unit (bin_id)
ON DELETE CASCADE,
    bin_class         integer    NOT NULL,
    date_time         timestamp(6) with time zone,
    variable_id         bigint                        NOT NULL  REFERENCES
observation_types (variable_id),
    variable_value      double precision              NOT NULL,
    position_date_time  timestamp(6) with time zone
);

CREATE TABLE data_profile
(
    submission_id     bigint    NOT NULL  REFERENCES submission (submission_id) ON
DELETE CASCADE,
    tag_id            bigint    NOT NULL,
    date_time         timestamp(6) with time zone,
    depth             double precision,
    variable_id         bigint                        NOT NULL  REFERENCES
observation_types (variable_id),
    variable_value      double precision              NOT NULL,    position_date_time
timestamp(6) with time zone
);
```

Appendix C – SQL Script for Running Materialized Views

Copy the SQL script below and execute it in PostgreSQL/Tagbase in the PgAmin console or save it to file as “tagbase-materialized-views.sql” and execute from the PostgreSQL command line.

tagbase-materialized-views.sql

```
-- Uncomment the line below if you run this from the terminal.
--\connect tagbase

-- MATERIALIZED VIEW

CREATE MATERIALIZED VIEW mview_vis_data
AS
 SELECT
    variable.submission_id AS source_id,
    variable.variable_value AS measurement_value,
    variable.variable_name AS measurement_name,
    variable.variable_units AS measurement_units,
    depth.depth,
    variable.date_time AS measurement_date_time,
    data_position.date_time AS position_date_time,
    data_position.lat,
    CASE WHEN data_position.lon > 180 THEN data_position.lon - 360 ELSE
data_position.lon END,
    data_position.lat_err,
    data_position.lon_err
   FROM ( SELECT x.variable_value,
            y.variable_name,
            x.date_time,
            x.submission_id,
            y.variable_units,
            x.position_date_time
           FROM data_time_series x,
            observation_types y
          WHERE x.variable_id = y.variable_id AND y.variable_name <> 'depth'
AND y.variable_name <> 'datetime') variable,
    data_position,
    ( SELECT x.variable_value AS depth,
            x.date_time,
            x.submission_id
           FROM data_time_series x,
            observation_types y
          WHERE x.variable_id = y.variable_id AND y.variable_name = 'depth')
depth
  WHERE variable.submission_id = data_position.submission_id AND
variable.submission_id = depth.submission_id AND variable.position_date_time
= data_position.date_time AND depth.date_time = variable.date_time
WITH DATA;


CREATE MATERIALIZED VIEW mview_vis_data_histogram
AS
 SELECT
    data.submission_id AS source_id,
    data.min_value AS bin_class,
    data.variable_value AS measurement_value,
    data.date_time AS measurement_date_time,
    data_position.date_time AS position_date_time,
    data_position.lat,
    CASE WHEN data_position.lon > 180 THEN data_position.lon - 360 ELSE
data_position.lon END,
    data_position.lat_err,
    data_position.lon_err
   FROM ( SELECT data_histogram_bin_info.min_value,
            data_histogram_bin_data.submission_id,
            data_histogram_bin_data.date_time,
            data_histogram_bin_data.variable_value,
            data_histogram_bin_data.position_date_time
           FROM data_histogram_bin_info,
            data_histogram_bin_data
          WHERE data_histogram_bin_info.bin_id =
data_histogram_bin_data.bin_id AND data_histogram_bin_info.bin_class =
data_histogram_bin_data.bin_class) data,
    data_position
  WHERE data.submission_id = data_position.submission_id AND
data.position_date_time = data_position.date_time
WITH DATA;



CREATE MATERIALIZED VIEW mview_vis_data_profile
AS
 SELECT
    data.submission_id AS source_id,
    data.depth,
    data.variable_value AS measurement_value,
    data.date_time AS measurement_date_time,
    data_position.date_time AS position_date_time,
    data_position.lat,
    CASE WHEN data_position.lon > 180 THEN data_position.lon - 360 ELSE
data_position.lon END,
    data_position.lat_err,
    data_position.lon_err
   FROM ( SELECT data_profile.submission_id,
            data_profile.date_time,
            data_profile.depth,
            data_profile.variable_value,
            data_profile.position_date_time
           FROM data_profile) data, data_position
  WHERE data.submission_id = data_position.submission_id AND
data.position_date_time = data_position.date_time
WITH DATA;


CREATE MATERIALIZED VIEW mview_vis_metadata
AS
 SELECT metadata.submission_id AS source_id,
    'Global Attributes'::text AS attribute_type,
    NULL::character varying AS variable,
    metadata_types.category,
    metadata_types.attribute_name,
    "left"("right"(metadata.attribute_value, length(metadata.attribute_value)
- 1), '-1'::integer) AS attribute_value
   FROM metadata_types,
    metadata
  WHERE metadata_types.attribute_id = metadata.attribute_id AND
(metadata_types.category::text = 'instrument'::text AND
(metadata_types.attribute_name::text = ANY
(ARRAY['instrument_name'::character varying, 'instrument_type'::character
varying, 'firmware'::character varying, 'manufacturer'::character varying,
'model'::character varying, 'owner_contact'::character varying,
'person_owner'::character varying, 'serial_number'::character
varying]::text[])) OR metadata_types.category::text = 'programming'::text AND
(metadata_types.attribute_name::text = ANY
(ARRAY['programming_report'::character varying,
'programming_software'::character varying]::text[])) OR
metadata_types.category::text = 'attachment'::text AND
metadata_types.attribute_name::text = 'attachment_method'::text OR
metadata_types.category::text = 'deployment'::text AND
(metadata_types.attribute_name::text = ANY
(ARRAY['geospatial_lat_start'::character varying,
'geospatial_lon_start'::character varying, 'person_tagger_capture'::character
varying, 'time_coverage_start'::character varying]::text[])) OR
metadata_types.category::text = 'animal'::text AND
(metadata_types.attribute_name::text = ANY
(ARRAY['condition_capture'::character varying, 'length_capture'::character
varying, 'length_method_capture'::character varying,
'length_type_capture'::character varying, 'length_unit_capture'::character
varying, 'platform'::character varying, 'taxonomic_serial_number'::character
varying]::text[])) OR metadata_types.category::text = 'end_of_mission'::text
AND (metadata_types.attribute_name::text = ANY
(ARRAY['time_coverage_end'::character varying, 'end_details'::character
varying, 'end_type'::character varying, 'geospatial_lat_end'::character
varying, 'geospatial_lon_end'::character varying]::text[])) OR
metadata_types.category::text = 'waypoints'::text AND
metadata_types.attribute_name::text = 'waypoints_source'::text OR
metadata_types.category::text = 'quality'::text AND
(metadata_types.attribute_name::text = ANY (ARRAY['found_problem'::character
varying, 'person_qc'::character varying]::text[])))
UNION
 SELECT data_time_series.submission_id AS source_id,
    'Variable Attributes'::text AS attribute_type,
    observation_types.standard_name AS variable,
    NULL::character varying AS category,
    'units'::character varying AS attribute_name,
    observation_types.variable_units AS attribute_value
   FROM observation_types,
    ( SELECT data_time_series_1.variable_id,
            data_time_series_1.submission_id
           FROM data_time_series data_time_series_1
          GROUP BY data_time_series_1.variable_id,
data_time_series_1.submission_id) data_time_series
  WHERE observation_types.standard_name IS NOT NULL AND
observation_types.variable_id = data_time_series.variable_id
UNION
 SELECT data_time_series.submission_id AS source_id,
    'Variable Attributes'::text AS attribute_type,
    observation_types.standard_name AS variable,
    NULL::character varying AS category,
    'standard_name'::character varying AS attribute_name,
    observation_types.standard_name AS attribute_value
   FROM observation_types,
    ( SELECT data_time_series_1.variable_id,
            data_time_series_1.submission_id
           FROM data_time_series data_time_series_1
          GROUP BY data_time_series_1.variable_id,
data_time_series_1.submission_id) data_time_series
  WHERE observation_types.standard_name IS NOT NULL AND
observation_types.variable_id = data_time_series.variable_id
UNION
 SELECT data_time_series.submission_id AS source_id,
    'Variable Attributes'::text AS attribute_type,
    observation_types.standard_name AS variable,
    NULL::character varying AS category,
    'long_name'::character varying AS attribute_name,
    observation_types.variable_name AS attribute_value
   FROM observation_types,
    ( SELECT data_time_series_1.variable_id,
            data_time_series_1.submission_id
           FROM data_time_series data_time_series_1
          GROUP BY data_time_series_1.variable_id,
data_time_series_1.submission_id) data_time_series
  WHERE observation_types.standard_name IS NOT NULL AND
observation_types.variable_id = data_time_series.variable_id
WITH DATA;
```

References

1. ACCESS15-0017 Oceanographic In-situ Interoperability Project Deliverable 1.2. TAG METADATA REVIEW & RECOMMENDATIONS DOCUMENT. Version: 1.0, February 27, 2017, 30p. https://oiip.jpl.nasa.gov/doc/OIIP_Deliverable1.2_TagMetadata_20170227.pdf